Compiler Construction
Assignment - 2

Documentation for Lexical Analyser and Parser

Name: Muhammad Osama Asif
Roll No: 17L-4295
Section: CS-8A

Lexical Analyzer

TOKENS

LEGEND:

- **Type** ➞ General kind of lexeme.
- **Family** ➞ Unique Identification.
- **Token** ➞ Family, lexeme pair.
- **Lexeme** ➞ The “word” taken from the source code.

NOTE: Where lexeme does not exist, **^** is used.

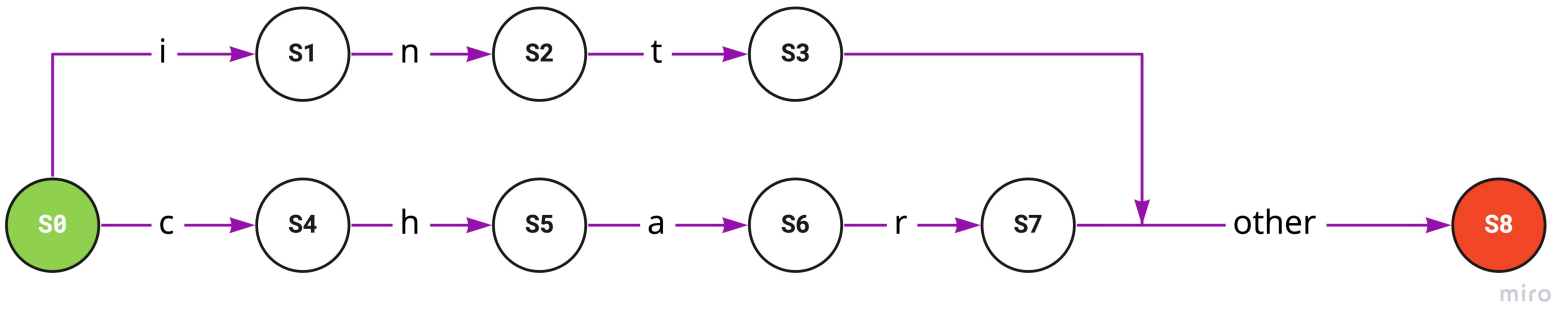
Type	Family	Token	Lexeme	Description
Data Type	Data Type (DT)	(DT, int)	int	Integer data type
Data Type	Data Type (DT)	(DT, char)	char	Character data type
Keyword		(WHILE, ^)	^	Used to loop through certain code
Keyword		(FUNC, ^)	^	Used to declare a function
Keyword		(RET, ^)	^	Used to return from a function
Keyword	Conditional Statement (CS)	(CS, if)	if	Used to run code based on condition
Keyword	Conditional Statement (CS)	(CS, elif)	elif	Used to run code based on condition
Keyword	Conditional Statement (CS)	(CS, else)	else	Used to run code based on condition
Keyword	Input/Output Statement (IOS)	(IOS, in)	in	Used to take user input
Keyword	Input/Output Statement (IOS)	(IOS, print)	print	Used to show (output) stuff
Keyword	Input/Output Statement (IOS)	(IOS, println)	println	Used to show (output) stuff
Operator	Arithmetic Operator (AO)	(AO, +)	+	Used to perform addition
Operator	Arithmetic Operator (AO)	(AO, -)	-	Used to perform subtraction
Operator	Arithmetic Operator (AO)	(AO, *)	*	Used to perform multiplication
Operator	Arithmetic Operator (AO)	(AO, /)	/	Used to perform division
Operator	Relational Operator (RO)	(RO, <)	<	Used to check if something is less than other
Operator	Relational Operator (RO)	(RO, <=)	<=	Used to check if something is less or equal to other
Operator	Relational Operator (RO)	(RO, >)	>	Used to check if something is greater than other
Operator	Relational Operator (RO)	(RO, >=)	>=	Used to check if something is greater or equal to other

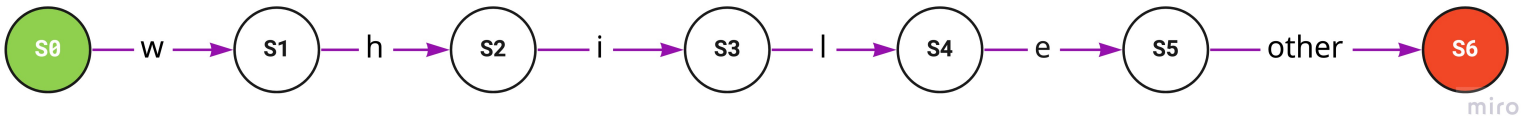
Operator	Relational Operator (RO)	(RO, =)	=	Used to check if two things are equal
Operator	Relational Operator (RO)	(RO, /=)	/=	Used to check if two things are not equal
Operator		(:=, ^)	^	Used to assign data to variable
Operator		(>>, ^)	^	Used to show where input data will be stored
Operator		(:, ^)	^	Used to declare a variable
Symbol	Comment (CMNT)	(CMNT, /*placeholder*/)	/*placeholder*/	Used to write comments inbetween
Symbol	Brackets (BRKT)	(BRKT, ( )	(	To wrap code
Symbol	Brackets (BRKT)	(BRKT,))	)	To wrap code
Symbol	Brackets (BRKT)	(BRKT, [ )	[	To wrap code
Symbol	Brackets (BRKT)	(BRKT,])	]	To wrap code
Symbol	Brackets (BRKT)	(BRKT, {)	{	To wrap code
Symbol	Brackets (BRKT)	(BRKT, })	}	To wrap code
Symbol	Punctuations (PUNC)	(PUNC, ,)	,	Used to separate items/ variables etc
Symbol	Punctuations (PUNC)	(PUNC, ;)	;	Used to mark end of a line of code
Symbol	Punctuations (PUNC)	(PUNC, :)	:	Used after while etc
Identifier	Identifier (ID)	(ID, a); (ID, x1) etc	a; x1 etc	Used for variable names
Literal	Numeric Constant (NUMC)	(NUMC, 1); (NUMC, 44) etc	1; 44 etc	Numbers
Literal	Literal Constant (LC)	(LC, ‘a’); (LC, ’X’) etc	‘a’, ‘X’ etc	Single Letters
Literal	String (STR)	(STR, “Hello”); (STR, “World”) etc	“Hello”; “World” etc	Group of letters etc

✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦ ✦

Regular Definitions and Transition Diagrams

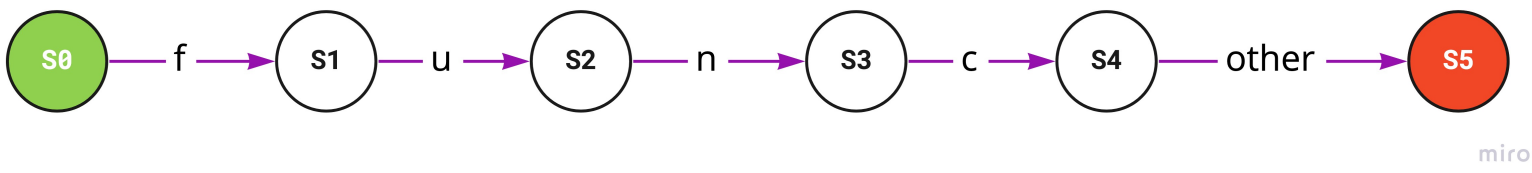
1) DT → int | char





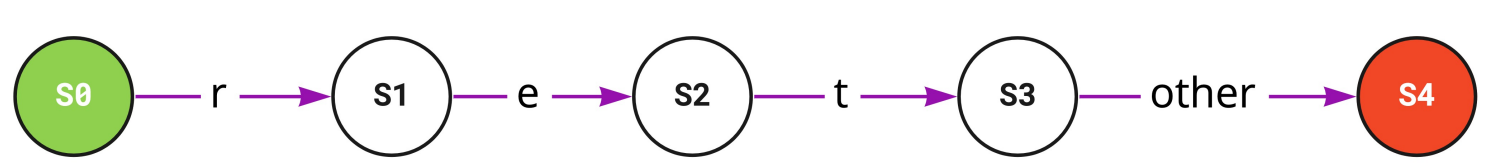
2) WHILE → while

3) FUNC → func



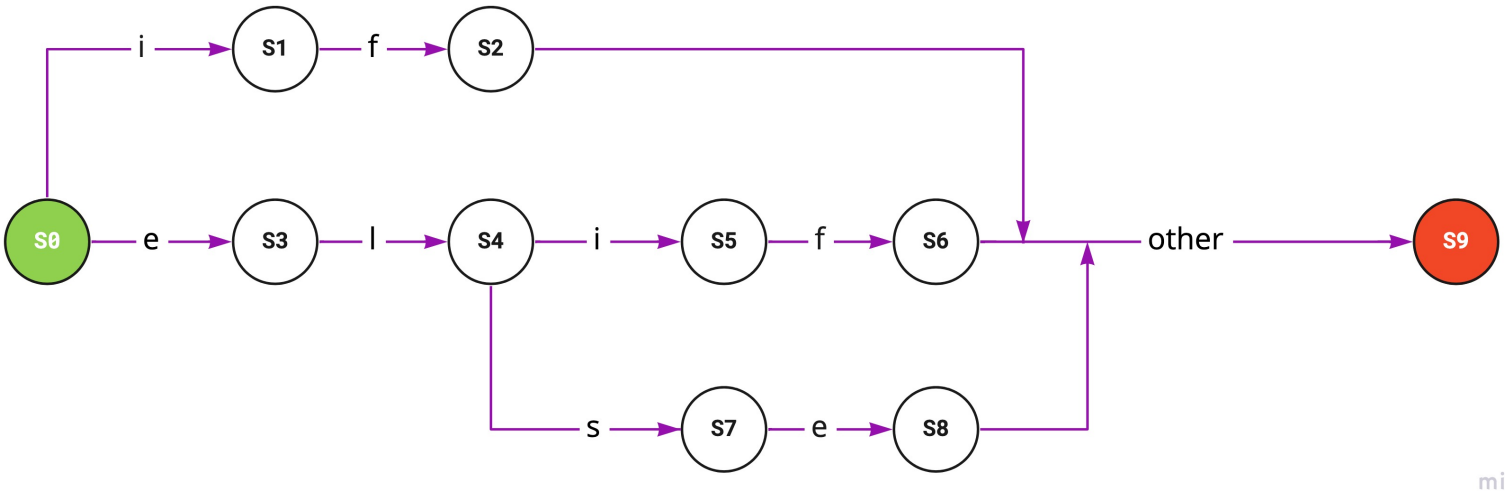
miro

4) RET → ret



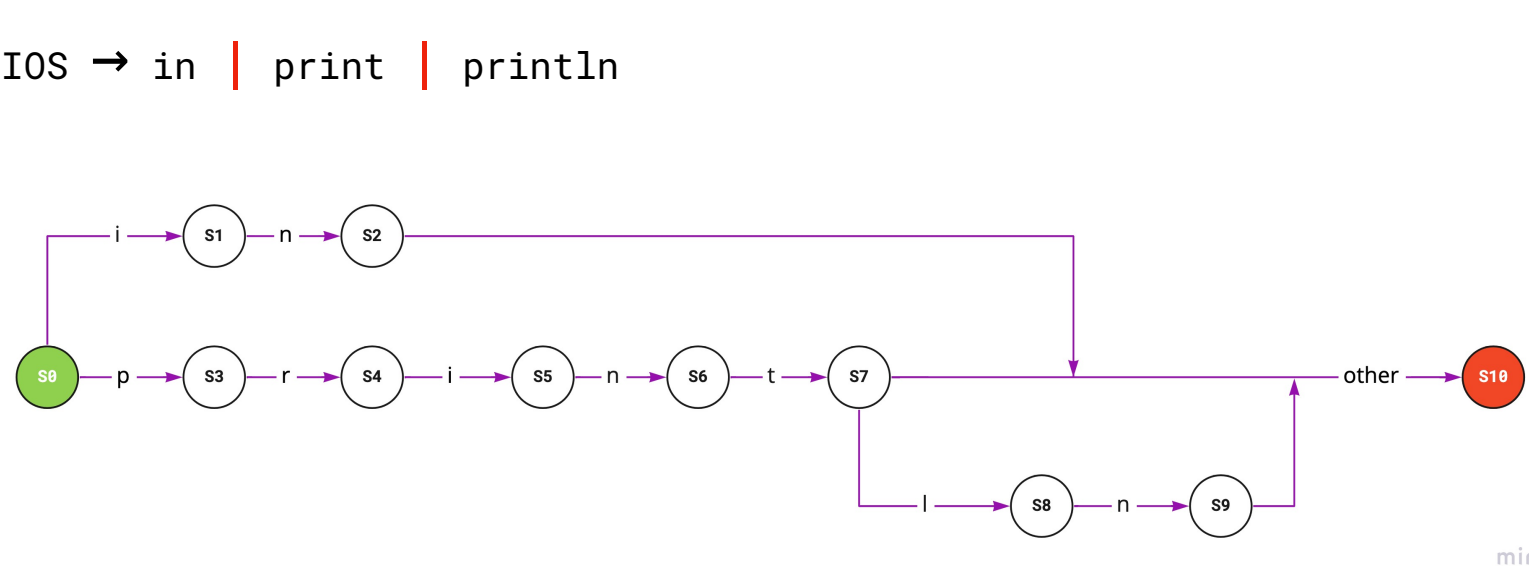
miro

5) CS → if | elif | else



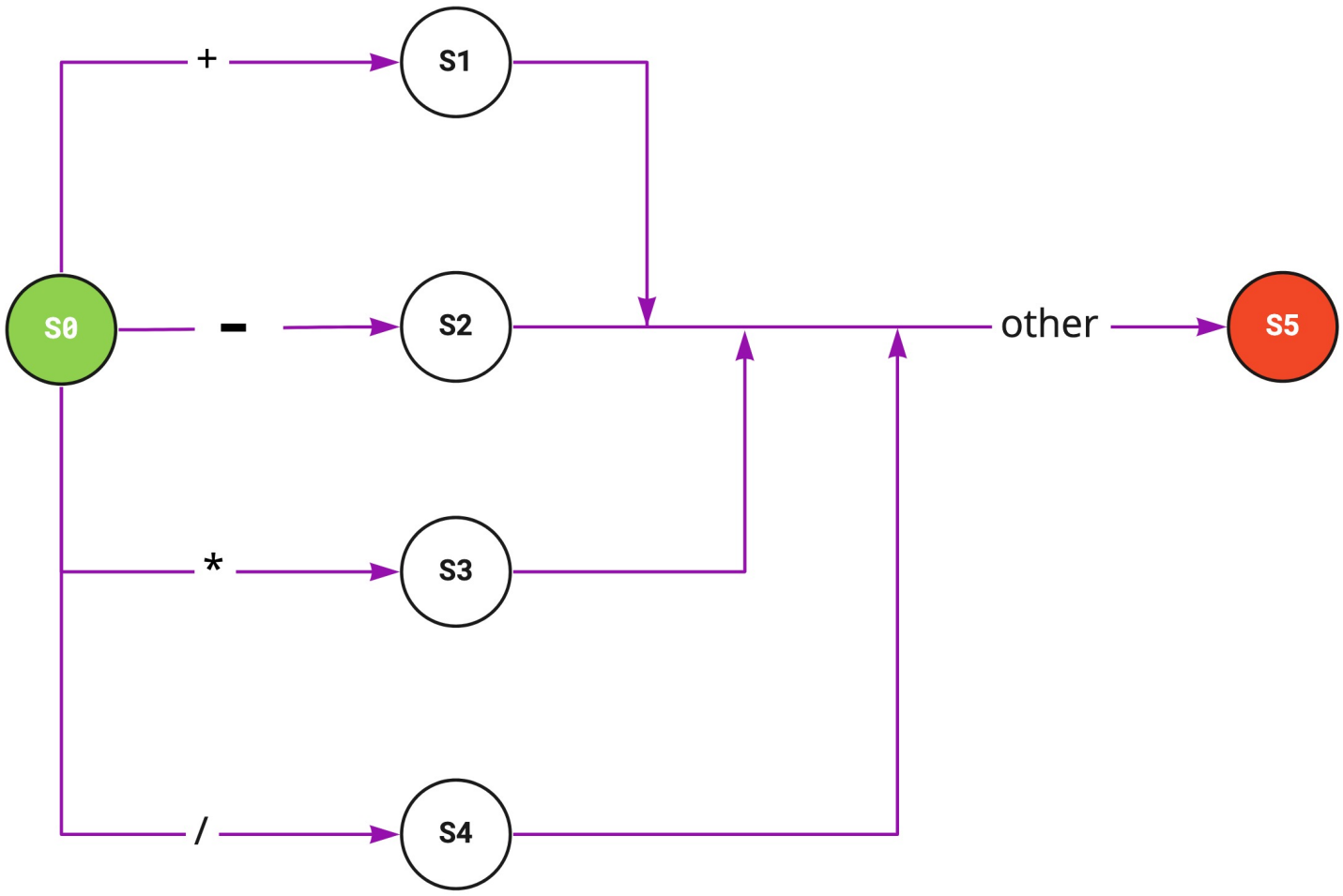
miro

6) IOS → in | print | println



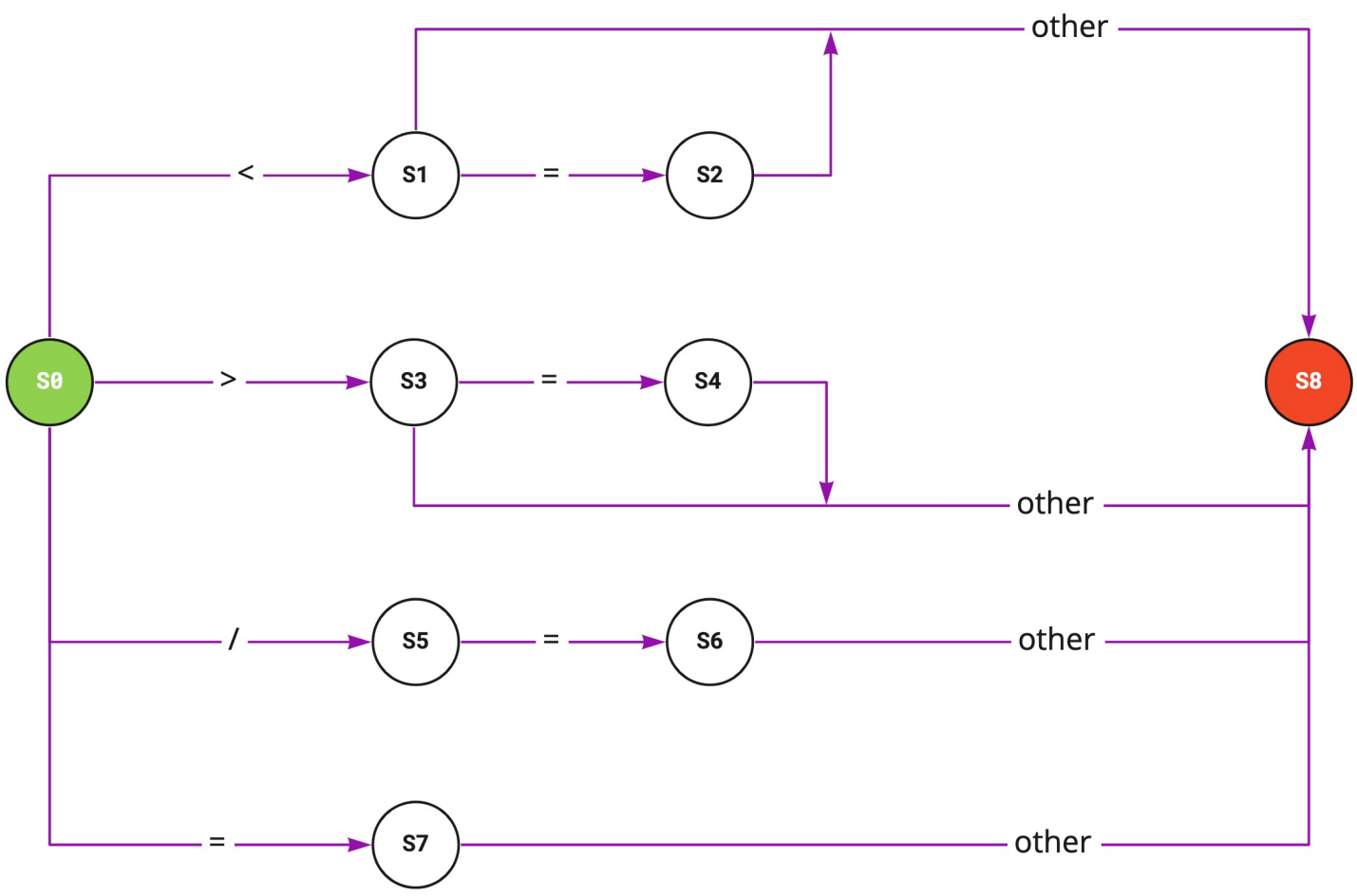
miro

7) AO → + | - | * | /



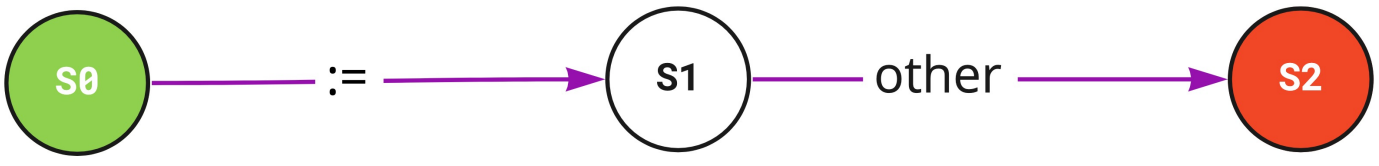
miro

8) RO → < | <= | > | >= | = | /=



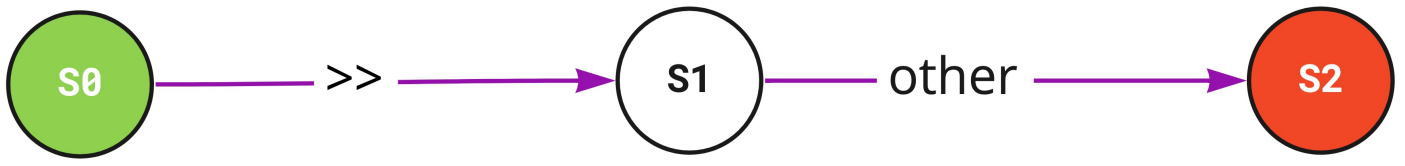
miro

9) := → :=



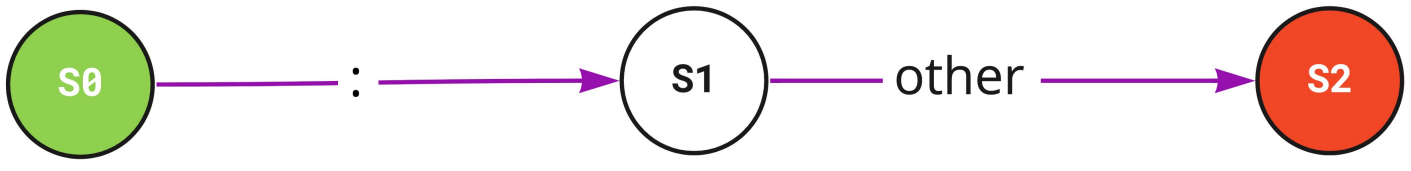
miro

10) >> → >>



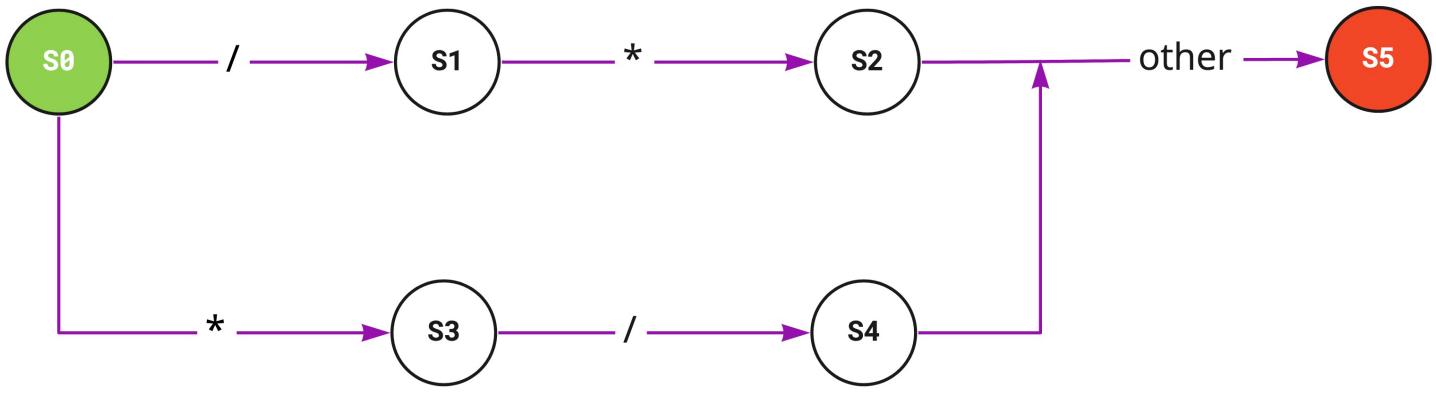
miro

11) : → :



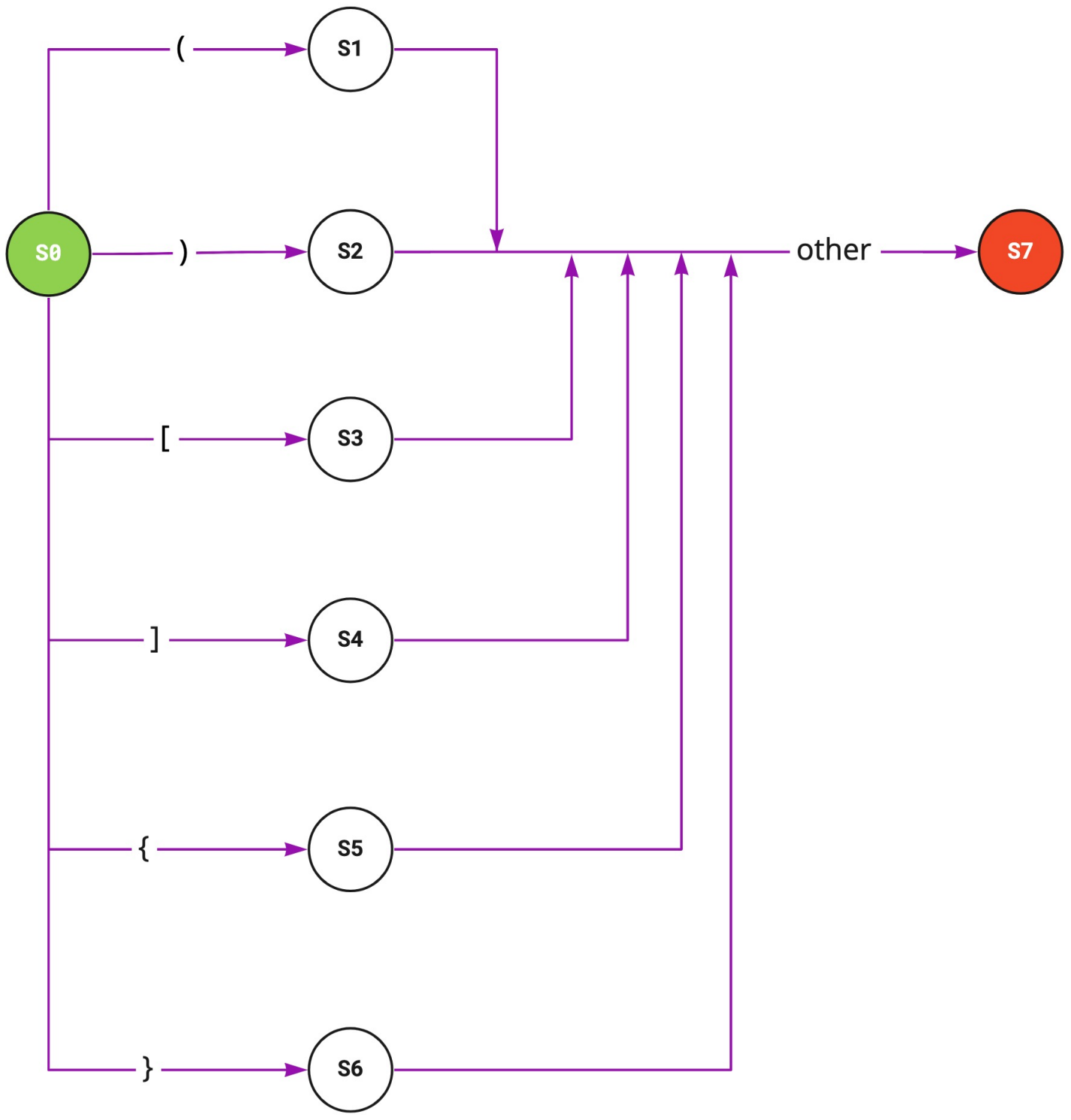
miro

12) CMNT → /* | */

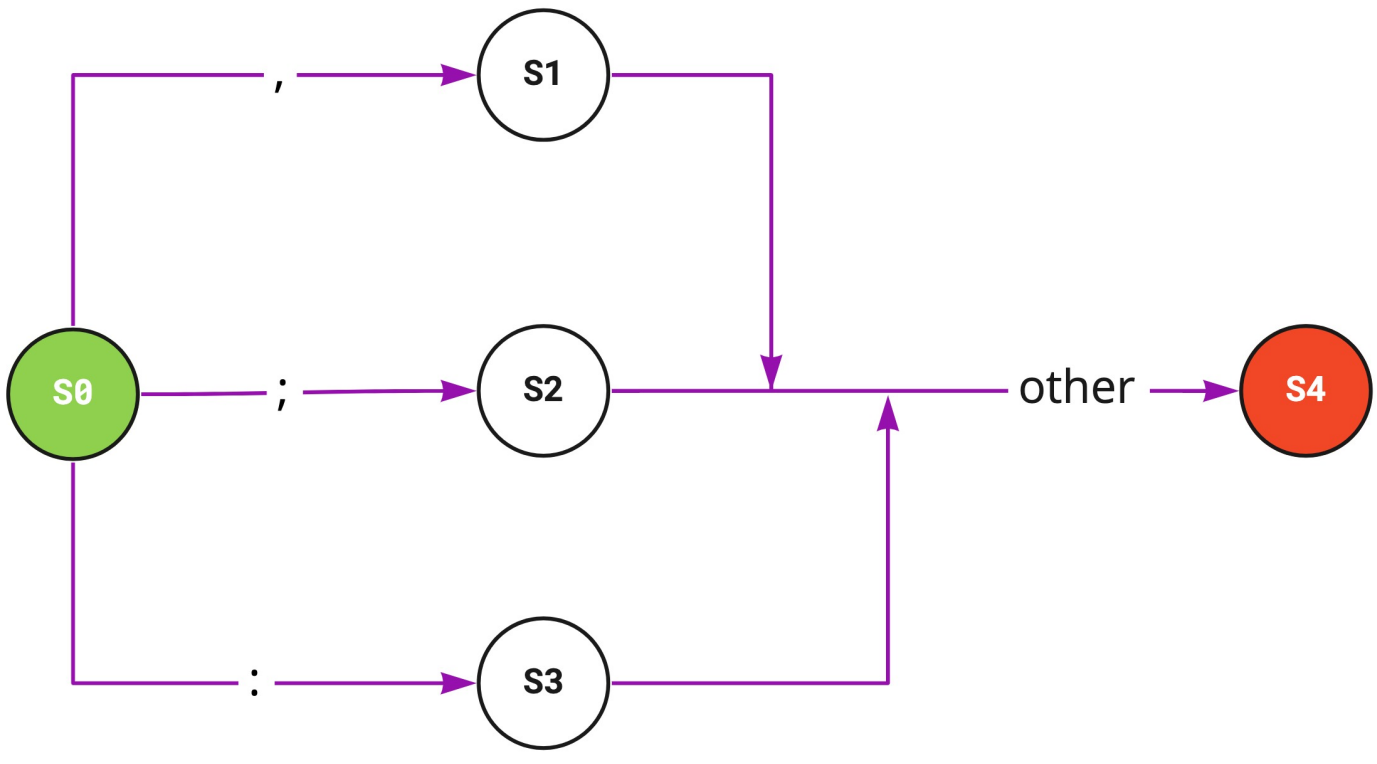


miro

13) BRKT → (|) | [|] | { | }



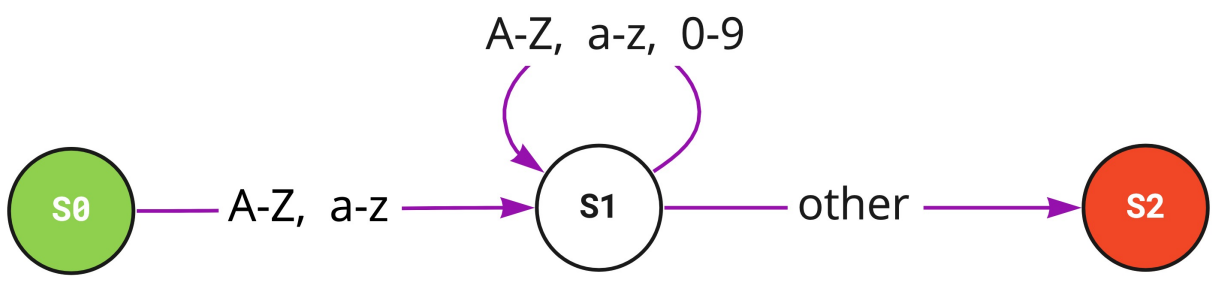
14) PUNC → , | ; | :



15) ID → LETTER (LETTER | DIGIT)*

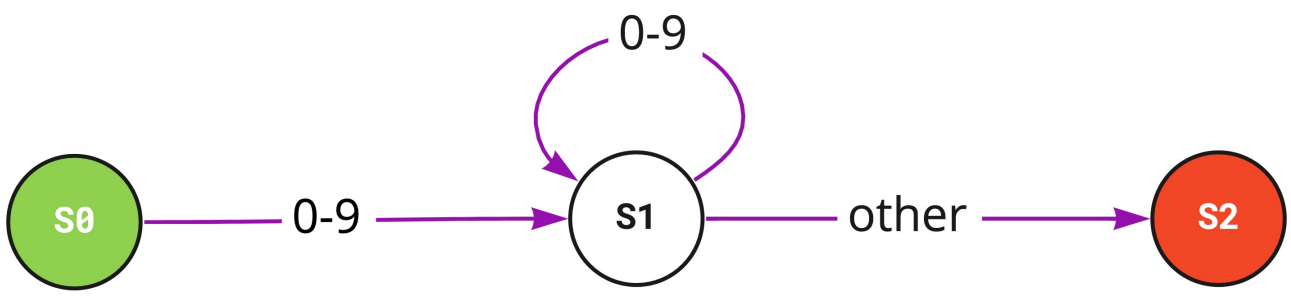
LETTER → [A-Z] | [a,z]

DIGIT → [0-9]



miro

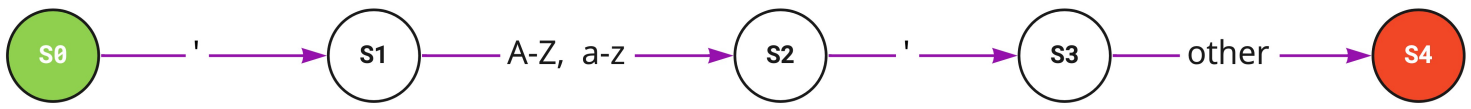
16) NUMC → [0-9]+ // + means one or more times



miro

17) LC → 'LETTER'

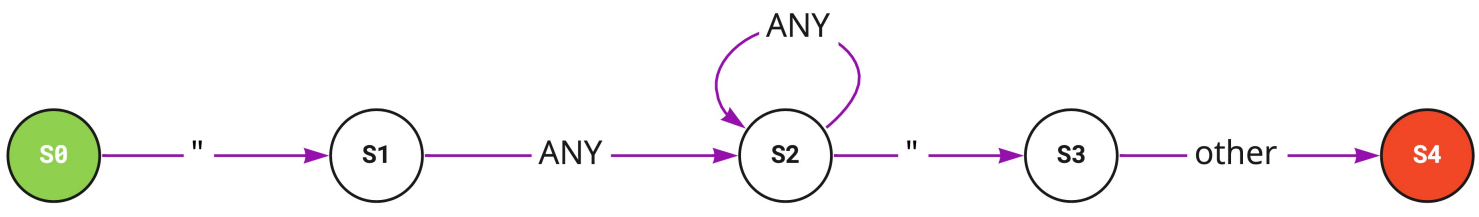
LETTER → [A-Z] | [a-z]



miro

18) STR → "ANY*" //dot(.) represents any in regex

ANY → [A-Z] | [a-z] | [0-9] | . | + | ? | $ | ? | ^ | * | (|) | [|] | { | } | | | \



miro

Parser

Context Free Grammar (CFG):

NOTE: Each CFG will have two steps;
1) Creation of the CFG.
2) Rewriting the CFG making sure it has left associativity, precedence, no left recursion, no ambiguity, and left factoring.

1) Code Block:

a) Initial CFG:

CODE → STMT ; CODE | ^

STMT → VARDEC | VARASSIGN | PR | IN | WL | CS | RET | W | AEXP

Note: FOR RIGHT SIDE OF **STMT** CHECK ALL THE CFG'S BELOW.

b) Rewritten:

Same as above.

2) Common ones used below:

a) DT → Integer | char

b) ID → ALPHA ID'

ID' → ID | DIGIT ID' | ^

ALPHA → [A - Z] | [a - z]

DIGIT → [0 - 9]

c) ANY → ALPHA ANY | DIGIT ANY | . ANY | + ANY | ? ANY | * ANY | $ ANY | ^ ANY | [ANY |] ANY | { ANY | } ANY | (ANY |) ANY | | ANY | \ ANY | ^

d) STR → " ANY "

3) Functions:

a) Initial CFG:

FN → **func** DT : ID (ARG) { CODE }

ARG → VAR ARG | , VAR ARG | ^

VAR → DT : ID

b) Rewritten:

Same as above.

4) Variable Declaration:

a) Initial CFG:

VARDEC → DT : VARDEC' ;

VARDEC’ → ID | ID , VARDEC’

b) Rewritten:

VARDEC → DT : VARDEC’ ;

VARDEC’ → ID VARDEC”

VARDEC” → ^ | , VARDEC’

5) Variable Assignment:

a) Initial CFG:

VARASSIGN → ID := VAL ;

VAL → ‘ ALPHA ‘ | ID

b) Rewritten:

Same as above.

6) Print Statements:

a) Initial CFG:

PR → **print (OUT) ; | println (OUT) ;**

OUT → STR | ID

b) Rewritten:

Same as above.

7) Input:

a) Initial CFG:

IN → **In >>** ID ;

b) Rewritten:

Same as above.

8) Loop:

a) Initial CFG:

WL → **while** CMP **: { CODE }**

CMP → ID RO WITH

RO → < | <= | > | >= | = | !=

WITH → ID | DIGIT

b) Rewritten:

Same as above.

9) Conditional Statements:

a) Initial CFG:

CS → **if** CMP **:** **{** CODE **}** CS’

CS’ → **if** CMP **:** **{** CODE **}** | **elif** CMP **:** **{** CODE **}** | **else** CMP **:** **{** CODE **}** |
^

CMP → ID RO WITH

RO → < | <= | > | >= | = | !=

WITH → ID | DIGIT

b) Rewritten:

Same as above.

10) Return Statement:

a) Initial CFG:

RET → **ret** ID **;**

b) Rewritten:

Same as above.

11) Write Statements:

a) Initial CFG:

W → **write** **(** OUT **)** **;**

OUT → STR | ID

b) Rewritten:

Same as above.

12) Arithmetic Expressions:

a) Initial CFG:

AEXP → ID **:=** E

E → E **+** T | E **-** T | T

T → T ***** F | T **/** F | F

F → ID | NUM | **(** E **)**

NUM → DIGIT NUM | ^

b) Rewritten:

AEXP → ID **:=** T E’

E’ → **+** T E’ | **-** T E’ | ^

T → F T’

```
T'      ➞   * F T' | / F T' | ^
F       ➞   ID | NUM | ( E )
NUM     ➞   DIGIT NUM | ^
```